Hosting an Object Heap on Manycore Hardware: An Exploration

David Ungar
IBM Research
dungar@us.ibm.com

Sam S. Adams
IBM Research
ssadams@us.ibm.com

Abstract

In order to construct a test-bed for investigating new programming paradigms for future “manycore” systems (i.e. those with at least a thousand cores), we are building a Smalltalk virtual machine that attempts to efficiently use a collection of 56-on-chip caches of 64KB each to host a multi-megabyte object heap. In addition to the cost of inter-core communication, two hardware characteristics influenced our design: the absence of hardware-provided cache-coherence, and the inability to move a single object from one core’s cache to another’s without changing its address. Our design relies on an object table, and the exploitation of a user-managed caching regime for read-mostly objects. At almost every stage of our process, we obtained measurements in order to guide the evolution of our system.

The architecture and performance characteristics of a manycore platform confound old intuitions by deviating from both traditional multicore systems and from distributed systems. The implementor confronts a wide variety of design choices, such as when to share address space, when to share memory as opposed to sending a message, and how to eke out the most performance from a memory system that is far more tightly integrated than a distributed system yet far less centralized than in a several-core system. Our system is far from complete, let alone optimal, but our experiences have helped us develop new intuitions needed to rise to the manycore software challenge.

Categories and Subject Descriptors D.3.4 [**Programming Languages**]: Processors - interpreters, memory management (garbage collection), Run-time environments: D.3.3 [**Programming Languages**]: Language Constructs and Features - Dynamic storage management, Concurrent programming structures : D.4.2 [**Operating Systems**]: Storage Management - Allocation/deallocation strategies: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors) - Multiple-instruction-stream, multiple-data-stream processors (MIMD): C.4 [**Performance of Systems**]: Design studies, Measurement techniques: C.3.2 [**Memory Structures**]: Design styles - Cache memories.

General Terms Design, Experimentation, Performance, Measurement.

Keywords *manycore, object heap, virtual machine, Smalltalk, Squeak, object table, cache performance*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DLS '09, October 26, 2009, Orlando, FL, USA.
Copyright © 2009 ACM 978-1-60558-769-1/09/10…$10.00

1. Introduction

The microprocessor industry’s transition from single core to multicore to manycore single chip multiprocessors poses an epochal challenge to the software industry. How can software applications reap continuing performance benefits from these disruptive improvements in hardware technology when most existing programming models and languages have co-evolved in the single core uniprocessor era? Despite decades of advances in computer science, programming parallel systems remains beyond the reach of all but a tiny handful of the world’s software developers. From petaflop computers with over one million cores like BlueGene/P™ [1], to Graphics Processing Units (GPUs) with hundreds of shader cores to the 9 core Cell/B.E.™ [2] in Playstation3™, programmability is widely recognized as a significant challenge.

The Renaissance project at IBM Research was chartered in 2008 as an exploratory research effort to take a clean-slate approach to finding a programming model suitable for the following twin challenges: how to fully exploit the massive parallelism of future manycore hardware in a manner that will also be accessible to the majority of today’s “productivity programmers.” We chose a two-pronged approach for this project, hardware-up and application-down. For hardware, we targeted a future hypothetical 1000-core heterogenous multiprocessor with a large majority of identical, general purpose cores. We are prototyping this system using 16 TILE64™ manycore processors from Tilera [3], each supporting 64 RISC-based processor cores. From the application perspective, we selected virtual world physics simulation to represent the class of large scale distributed relaxation problems we

believe will finally be commercially feasible on future manycore systems.

We selected the Squeak Smalltalk system [4] as a software substrate for our investigations. As a widely ported open source virtual machine, class library and full development environment, Squeak was one of the most flexible systems available. The authors' extensive experience in Smalltalk virtual machine design, application development and programmer education was a big plus, and the tiny size of the system relative to alternatives like commercial Java systems was a good fit for the smaller, relatively slow and cache-starved cores of our chosen hardware.

Having selected our hardware and software starting points, the next step was to port the existing Squeak Smalltalk virtual machine to a single TILE64 manycore processor and rewrite it to function in parallel on the 56 cores available for user programs. (The other 8 cores were running the Linux operating system device drivers in the configuration we were using.)

In moving from a single-core Squeak virtual machine, to a 56-core system, we decided to tackle the memory system first, and save application-level parallelism for a subsequent effort. The memory system can be further subdivided into support for reclaiming unused memory (garbage collection), and support for creating, accessing, and modifying objects (the mutator). While garbage collection presents an interesting challenge for this system, we focused on the mutator portion of the memory system and deferred advanced work on the collector.

This paper describes:

- a series of experiments we undertook to figure out how to host an object heap on manycore hardware,
- the evolution of the resulting design, and
- its effectiveness at overcoming the performance penalty of dispersing the heap over a large number of small caches.

The roadmap for this paper is as follows: We first discuss multiprocessor Smalltalk/Java systems, then overview our chosen hardware and software platforms, and finally describe our efforts to implement a manycore object heap in detail.

2. Related Work: Multiprocessor Smalltalk/Java Systems

Space limitations force us to omit discussion of much important work, especially in the areas of scientific computing, distributed systems, ultra-large systems, and even related manycore systems.

MS was, as far as we know, the first system to run a Smalltalk or Java-like system on a multiprocessor [5, 6]. The hardware was a five-processor DEC Firefly, a machine with five MicroVAX processors and 16 Mb of shared memory, with a per CPU cache of 16 Kb, and hardware support for cache coherency [7]. The virtual machine was an interpreter, based on Berkeley Smalltalk [8] and the memory system was based on a later version of Berkeley Smalltalk that incorporated Generation Scavenging and eliminated indirection (via an object table) for object references [9, 10].

The J-machine was a distributed system built in anticipation of high levels of on-chip integration [11]. It ran a language called Concurrent Smalltalk, a class-based language with some static typing, parallel execution semantics, and a Scheme-like syntax [12]. It used an object table (called a BRAT), but rather than putting that table in globally-accessible memory, it relied on sending messages to an object's "home" processor in order to find the object.

The Mushroom system was a multiprocessor designed for Smalltalk [13, 14]. Its memory system hardware mapped virtual to physical addresses at a fine granularity. This design permitted the runtime system to move an object around in physical memory without paying a speed penalty for address translation [15]. The Mushroom memory architecture was the inspiration for our choice to employ an object table as a means to move an object from core to core. Unlike Mushroom, our mutator must pay a time penalty as the address translation is performed by software.

3. The TILE64 Manycore Processor

The TILE64 processor, based on the Raw machine project at MIT [16], may well portend manycore computer architectures to come. Containing a grid of 64 RISCs, its design emphasizes scalability. Each tile includes a network processor in addition to the instruction processor.

The designers of manycore processors must trade off the number of cores against the amount of memory local to each core. As the local memory size increases, the area occupied by each core increases, and given a constraint on the size of the die the number of cores decreases. We believe that future manycore processors may likely have far less per-core memory than today's multicore CPUs. The TILE64 processor seems like a harbinger of things to come in this respect as well, albeit perhaps an extreme data-point.

As Table 1 shows, each Tilera core possesses a mere 64 KB of local memory (known as the L2 cache), and a remote (inter-tile, or main memory) access will require a wait of at least 35 cycles. If you are old enough, you might think of this chip as a miniature roomful of incredibly fast PDP-11's. The last column in Table 1 illustrates the criticality of cache hit rates by estimating the MIPS that would result from code with extremely poor caching behavior.

In concert with the hardware itself, Tilera provides an extensive environment to support software development for the TILE64: The Tilera Multicore Development Environment [17], or MDE, provides numerous command-line and Eclipse-based tools for developing, debugging, and profiling manycore applications.

condition	cache size, bytes	miss penalty, cycles	line size, bytes	worst case MIPS equivalent
max		0		600 - 1800
branch mis-predict		2		250
data in L1	8 KbI, 8 KbD	2	16 b	250
miss L1, hit L2	64 Kb	8	64 b	80
miss L2, hit L3	4 Mb	35–49	64 b	20
miss L3	4 Gb	69–88		10

Table 1. TILE64 cache characteristics

The TILE64 was originally targeted toward streamed video compression, packet filtering and other network applications. Our project, on the other hand, explores how to run general-purpose object-oriented programs on such an architecture. Since such a system relies on a heap of interconnected objects ranging from a few to hundreds of mega-bytes, we believe that minimizing the overhead of dispersing such a heap across a myriad of cores will be critical to obtaining reasonable performance. In this paper, we report on our experiences in addressing this problem.

4. Squeak

Our work is based on the Squeak open source Smalltalk system [18]. This system includes support for the Smalltalk programming language, an integrated development environment (IDE), and a virtual machine based on a bytecode interpreter. The language includes features such as user-defined control structures and operator overloading that will help us explore new parallel programming paradigms. The IDE is tuned toward an exploratory style of programming that will also expedite our research and foster creativity. Since the Squeak system is self-hosting, i.e. the tools and environment are written in Smalltalk, every time we use the environment we are testing our virtual machine.

In addition, Smalltalk's age yields another benefit: The older of its two user interface frameworks (known as MVC, the original implementation of Model-View-Controller [19-21]) was originally designed in the late 1970's, long before the age of gigabyte memories and gigahertz processors. As a result, it reflects compromises between generality and efficiency that permit a machine that ran almost impossibly slow by 2009 standards (a 2 MB Macintosh Plus from 1986, for instance) to provide a good interactive graphical experience while running in an interpreted language. By contrast, newer systems built for languages such as Java have expanded to consume the bountiful resources of modern pre-manycore hardware.

Of all of Squeak's components, its virtual machine is most central to this effort. As a result of the optimizations developed by the original Smalltalk implementors [18], a virtual image containing the full IDE, two user interface frameworks, a generous class library (both source code and compiled methods) and a plethora of auxiliary tools fits in 16 MB, and a stripped-down image containing a fully-functional IDE and class library can fit in as little as 1.7 MB. These numbers are dwarfed by today's mainstream Eclipse environments and Java heaps. By using Smalltalk, we get to simulate a future of larger (mainstream) heaps running on a manycore processor with larger per-core memories.

Smalltalk's memory efficiency partly results from its reliance on a bytecode interpreter for execution, rather than a (dynamic) compiler. Future manycore object-oriented virtual machines may well rely on compilation, but for our work, we elected to save both implementation effort and possibly cache misses by sticking with an interpreter. As it turned out, misses in the instruction cache still accounted for a substantial number of stalls (see section 5.4). Readers who wish to use our results in guiding their own compilation-based designs will have to account for the consequences of a different execution mechanism.

The Squeak virtual machine represents activation records ("contexts") by ordinary Smalltalk objects. The currently executing context is called the "active context." Squeak also implements multiple, cooperatively-scheduled threads that all run in the same object heap. Although the use of contexts often adds overhead to calls and returns when compared to using frames on a stack, we felt that the potential ease of thread migration via simply migrating context objects outweighed the performance penalty. Our virtual machine therefore follows the classic Squeak/Smalltalk model by using context objects for its activation records.

Although we benefited from the current design of the Squeak virtual machine, we knew that the memory system would be substantially different and hence, guided by the reference implementation, rewrote it and the bytecode interpreter. We reused the existing code for some primitives, including the graphics operations, that were written in C in the original Squeak system. In our rewritten portions, we used the C++ static type system to ensure that object references were not conflated with object addresses. The rest of the paper explains the changes we made and our experience with them.

5. Our Experience

Upon the arrival of our Tilera processors and host computers, we undertook to gain insight into our platform and then port Squeak to it.

5.1. TILE64 Measurements and Characteristics

In order to guide our efforts, we undertook to measure the performance of various operations on the TILE64 processor. Our measurements used the cycle counter of the processor. (We believe that there is an uncertainty of a cycle or two, stemming from the details of instruction execution.) The measurements confirmed Tilera's figures about the relative speeds of the three levels of channel-based messages: raw, streaming, and buffered. As expected, raw messages were the fastest, taking twenty-three cycles to send, and twelve to receive a 10-word message.

5.1.1. Communication via the Memory System

In addition to messaging, the TILE64 processor supports direct-memory-access (DMA) between main memory and cache, or between different tiles' caches. DMA provides the benefit of parallel data transfer and instruction execution. We compared the time (in cycles) for various DMA operations with simple instruction loops to transfer data. When transferring data from one tile to another, we measured two distances: adjacent tiles (i.e. one hop), and tiles separated by two intervening tiles (i.e. three hops):

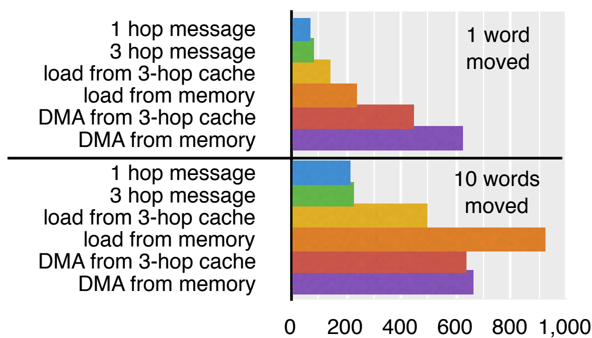

Figure 1. TILE64 data movement times (cycles)

As expected, hops added time, and DMA transfers were uneconomical for 10-word messages. Perhaps surprisingly to those who, like us, are new to manycore systems, raw channel messages transferred data faster than executing load/store instructions. The bottom line is that for short transfers, programmed message-passing saves time compared to either programmed- or direct- memory access.

Space limitations preclude further discussion, but our results agreed with the information provided by Tilera.

5.1.2. Homing

It was clear at this point that, as expected, object location would be critical thus we assumed that the virtual machine would need to be able to relocate an object from one core to another. Therefore, we needed to understand the caching policies for our hardware. The TILE64 processor supports three caching regimes:

- *Read-only* data may be cached on any tile, as needed.
- *Read-write* data may only be cached on one specific tile, determined by the page containing the data. If read or written by another tile, each word imposes a latency of about 35 cycles plus 2 cycles per hop. The caching tile is called the "home" tile, and this mechanism is called "homing."
- *User-managed* data may be cached on multiple tiles, but before performing a store instruction, all other tiles must invalidate cache lines for the data, and after performing the store, the storing tile must force the cache line out to main memory.

5.1.3. Hardware vs. Software Memory Management

As we thought about the application of the TILE64 architecture to the problem of hosting a heap of objects, we realized something that had a profound effect on the design of our virtual machine's memory system. Our IBM colleagues David Bacon and David Grove, both expert in Java virtual machines, reminded us that for the sake of efficiency it was best to get the hardware to do as much work as possible. That led us to the decision to let the hardware fill and flush the cache on each core, rather than use software to intervene. As a consequence, we employed the *read-write* mode, a decision we would later revisit. Thus, we planned to divide the objects up into N contiguous spaces (which we perhaps confusingly called "heaps"), one per core, and let the hardware control the movement of data from heaps in main memory to caches.

We also wanted to be able to relocate an object from one core to another, in the hope that we could minimize access time by maximizing locality. This relocation requirement meant that we would need to change the assignment of an object to a cache. However, in *read-write* mode, the TILE64 assigns all data in a given page to the same core. Since the average object at 44 bytes is far smaller than the smallest page at 64 KB, and given the scarcity of translation look-aside buffer (TLB) entries, it would not have been practical to relocate an individual object to another core without software intervention.

5.1.4. Summary of Critical TILE64 Characteristics

Putting it all together:

- Accessing an object that is not cached locally takes a lot of time.
- Local memory is a critical resource.
- The most efficient method for moving an object or performing a remote procedure call employs raw channels to pass messages.
- Raw channels require the software to perform flow control.
- Scanning the local cache is prohibitively expensive.
- The minimum page size of the TILE64 hardware combined with the number of TLB entries rule out hardware-based object location management. Subsequent designs for Tilera manycore processors have tackled this problem [22].

With these lessons in mind, we set about our initial design for the memory system.

5.2. Basic Memory System Design

The standard Squeak system featured direct pointers; an object reference was the same as its address:

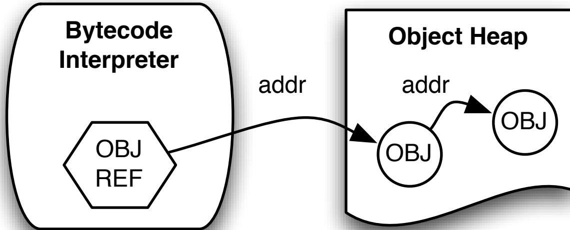

Figure 2. Direct object pointers

Given that object location is critical, and given the lack of hardware support for object-granularity homing, we decided to travel back in time to 1980 and implement an object table:

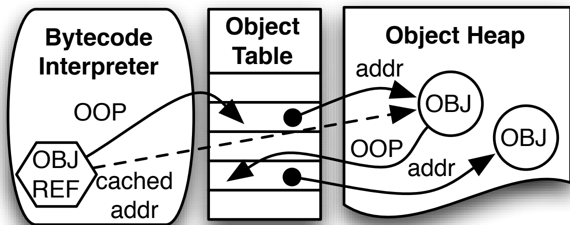

Figure 3. Object Table

An object reference is the address of an Object Table Entry (OTE), which in turn contains the address of the object. This indirection through the object table permits our system to move an object to a different core by first copying the contents of the object to another page and then updating its OTE. In order to be able to find an object's OTE from the object, we also added a backpointer word to the object's header.

There are disadvantages to this scheme: while moving an object other cores may not write to the object; object reference traversals are slower; there is no good single place in which to cache the object table; object table entries must be reclaimed; object table compaction is problematic; and there is a 10% space penalty for the extra header word. The drawbacks were outweighed by this scheme's ease of implementation, which allowed us to experiment with dynamic object relocation earlier than other alternatives.

Wanting to let the hardware do as much as possible, we divided the object space into multiple heaps, one heap per core. Each heap was the same size, a power of two, as well as an integral number of pages. This constraint optimized the computation of the home rank of an object from its address. Each heap was described by a "Heap" data structure containing a start, a next-free, and an end pointer. Each heap, along with its associated "Heap" structure was homed to its owning core. We used a shared address space so that any core could access any object in any heap, or any "Heap" structure:

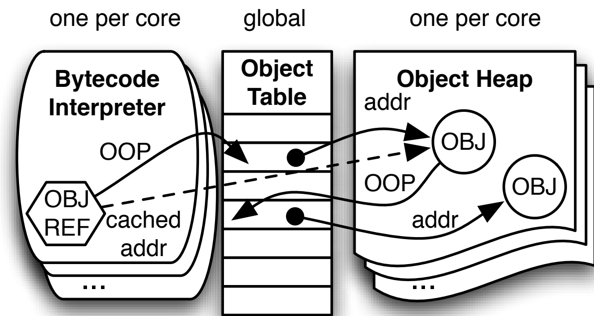

Figure 4. Multiple interpreters, global Object Table, multiple heaps

Throughout this project we found ourselves juggling decisions about shared vs. private memory, shared vs. private addresses, and homing considerations. These sorts of design decisions seem to be a characteristic of the manycore world, in which the costs of sharing significantly exceed those of the current few-core processors, yet remain economical in many cases, unlike the world of distributed systems.

A Smalltalk virtual machine reads in a large number of objects from a binary file, called a "snapshot," at startup time. Our system distributed these objects to the various heaps in round-robin fashion, like a dealer dispensing cards. After reading in these objects, the interpreter commenced execution. During execution, when a core allocated an object, it placed it in its own heap. Thus, the store operations required to update the "Heap" structure and to initialize the object were local.

In order to find the Heap structures, each core had its own (immutable) array of pointers to them:

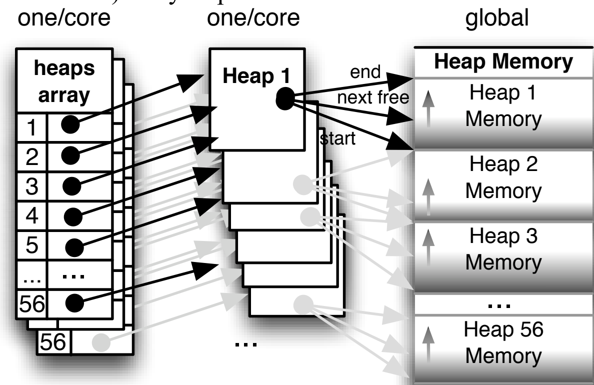

Figure 5. Heap Array and Heap Memory Layout

In this phase of our project, we wanted to optimize memory accesses while deferring work on application-level multithreading. Therefore, we chose to implement an interpreter with only a single thread and to have that thread hop around from core to core in order to optimize memory references to objects. (In Smalltalk, "sending a message" to an object is equivalent to a virtual function call in other

languages.) We expected the most frequently accessed objects to be the active context and the message receiver. So, in order to maximize locality:

- When sending a message to an object the new active context was allocated on the same core as the receiver's home.
- On every call or return the interpreter thread hopped to the core hosting the receiver object of that activation.

Consequently, references to the receiver and active context were always local to the core running the interpreter. We called this thread hopping "passing the baton" from core to core. The simple algorithm shown below later increased in complexity as our design evolved:

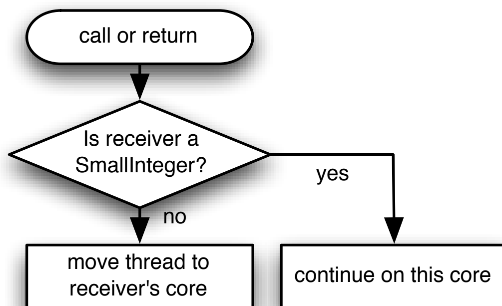

Figure 6. Basic baton passing algorithm

Although these mechanisms ensured local accesses for the receiver and active context, accesses to other objects, including message arguments would likely be remote, i.e. require communication with other cores' caches or main memory. Our virtual machine used the memory system for these, although messaging could potentially have been faster. When we eventually run multithreaded applications the memory system may be the more efficient option.

Recall that we reused existing C code for primitive operations such as graphics and file operations. In general, this code depended on local state, so it was necessary to run these primitives on the same core that had originally started up and initialized the virtual machine (the "main core"). Thus, whenever one of these foreign primitives was invoked, our system passed the baton back to the main core for the duration of that primitive.

5.3. Summary: Contrasts with the single-core virtual machine

In moving from a single-core to our first manycore object memory:

- We introduced a level of indirection between an object reference and the object's address,
- We divided up the heap into contiguously-addressed individual heaps sized in multiples of pages and powers of two,
- We distributed objects in the snapshot to each heap in a round-robin fashion,
- We deferred multithreading of the mutator, instead passing the baton from core to core,
- New objects were allocated in the heap of the core currently running the interpreter,
- We ensured that the active context and receiver objects were always local to the core running the interpreter,
- Primitives that we did not rewrite were always executed back on the "main" core, by passing the baton back and forth,
- We provided primitive operations to allow the Smalltalk-level code to move objects from core to core,
- We provided primitives to support visualization tools in Smalltalk (with apologies to Heisenberg, see Figure 7),
- We implemented the simplest possible garbage collector in order to avoid redoing others' research.

After implementing these ideas, it was time to find out how well they worked.

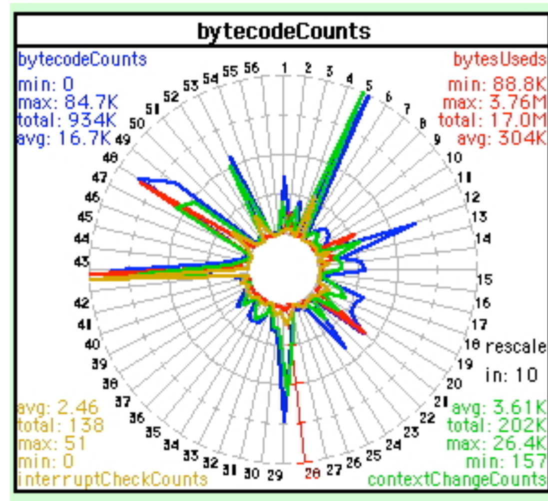

Figure 7. Real-time monitoring within the manycore Squeak environment

5.4. Evaluation of Initial Design

How often in this field does one get to return to a twenty-year-old benchmark? In the early 1980's, the Smalltalk "Compiler Benchmark," which runs the Smalltalk program that parses a method, emits bytecodes, and creates a method object, was the gold standard for Smalltalk systems. Ungar relied on it when he implemented the first Smalltalk system without an object table and with 32-bit object references , and when he evaluated the performance of a special-purpose RISC for Smalltalk [10]. Twenty-six years later, it was time to run this benchmark again. It was a bit like a reunion with an old friend.

The Smalltalk tradition is to measure the real time elapsed for a benchmark, and since our TILE64 processor was only running our virtual machine, it seemed reasonable to follow that tradition.

For our tests, we could confine the virtual machine to any number of cores from one to 56. The other eight cores were running the Linux operating system device drivers in the configuration we were using.

5.4.1. Mystery: double the cores, a third of the performance

Since our system was only running one interpreter thread at a time, while passing the baton among the cores, we expected the time taken to run the benchmark on two cores to be only slightly longer than the time to run the benchmark on one core. Instead, the time tripled going from single- to dual-core! Worse still, it almost doubled again from two-cores to fifty-six cores. (See Figure 8.)

In order to investigate this mystery, we first tried the profiling facility in the Tilera Multicore Development Environment. The profiler output, which is based on hardware counters, reported overall statistics. Needing more detailed information, we turned to the Tilera cycle-accurate simulator. Since simulation was much slower than execution, we had to implement a checkpointing feature in our virtual machine, so we ran at full speed to load in a snapshot, converted all of the data, and then checkpointed the virtual machine state. The simulator would then run our checkpoint restoration code, and proceed to the benchmark. Each core simulated took about twenty minutes to run our short benchmark, so we focused on comparing single- to dual-core configurations. The very first experiment counted bytecode dispatches in order to verify that the dual-core system was not executing more Smalltalk code than the single-core system. The TILE64 CPU combines two or three instructions into a single bundle, and we also verified that there was no difference in bundling efficiency.

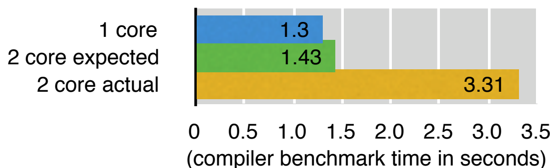

Figure 8. Mystery: Our code runs far slower on two cores than on one.

To narrow down the causes of the mysterious performance penalty when running dual-core, we started by investigating whether the unexpected overhead was caused by executing extra instructions, or by stalling the instruction pipeline more than before:

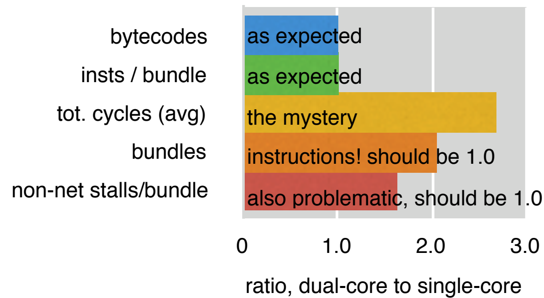

Figure 9. Are stalls or instructions the problem?

The extra total cycles (ratio of 2.7) confirmed that the simulator was running our benchmark and reproducing the time difference observed in the real system. The ratio of bundles retired was about 2.0, suggesting that the system was indeed executing extra instructions – twice as many! – in the dual core case. We then factored out stalls caused by inter-core reads, since only one core should be running at a time, and then compared the stalls per bundle. That ratio was 1.5, suggesting that there was also a problem with less-efficient instruction execution. Our mysterious overhead was caused by both instructions and stalls!

The reason for executing extra instructions was not obvious from the profile information, so we looked at the (non-network) stalls:

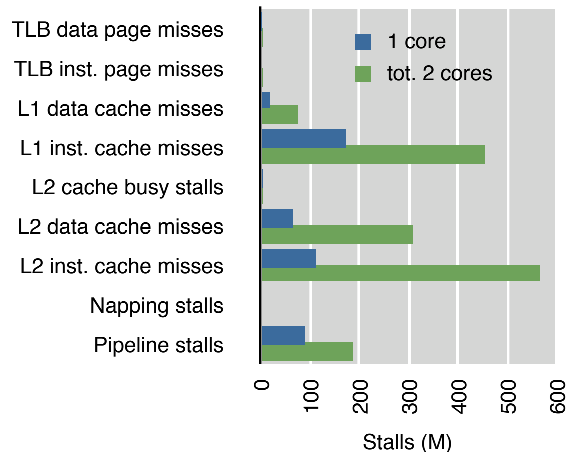

Figure 10. Stalls: dual- vs. single- core

Although both instruction and data stalls showed up as problems, when we looked at the causes of the L2 data stalls the mystery was solved:

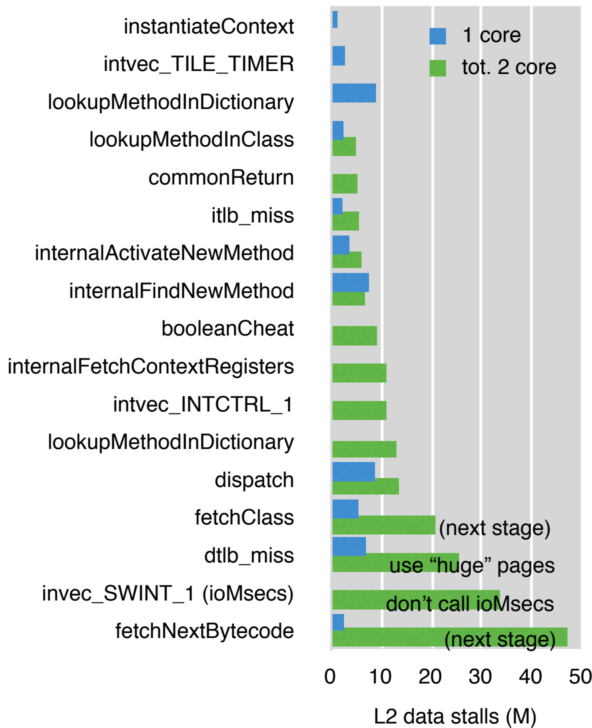

Figure 11. Source of L2 Data Stalls

The largest increase in L2 data stalls was caused by the interpreter reading bytecodes from methods, as indicated by the bar for the "fetchNextBytecode" routine. Recall that our heuristics ensured that the active context and receiver would be local, but did not consider the location of the method objects themselves. This was a harder problem to attack, so we deferred it (see section 5.5).

The second-largest increase was in a runtime routine ("invec_SWINT_1") used by the real-time clock system call invoked from a routine in our virtual machine ("ioMsecs"). We had instrumented our system with code to track the real time and cycle count of every migration of the interpreter thread to another core. No wonder that when running on two cores, the real time clock was queried far more than on the single core configuration! Since the cycle counter was providing the same information at a much lower cost, we fixed this problem by simply removing the calls to "ioMsecs" in our instrumentation.

The third-largest increase in L2 data cache stalls occurred in a routine that was handling misses in the data page translation look-aside buffer (DTLB). The standard TILE64 page size is 64 KB, and each core has only 16 entries for data pages (eight for instruction pages), another difference resulting from the large number of smaller capacity cores. With a heap size in the range of ten megabytes, it made sense that there would be many of these misses. We attacked this problem by switching to an optional large page size of 16 MB, called "huge pages" in this system.

The fourth-largest increase in L2 data cache stalls occurred in the "fetchClass" routine. This routine is most often invoked when a message is sent to an object in order to lookup the method based on the object's class. This issue was also deferred.

At this point, we had tackled the second and third place culprits of L2 data cache stalls, and deferred the 1st and 4th place culprits. We measured the effects of the two fixes:

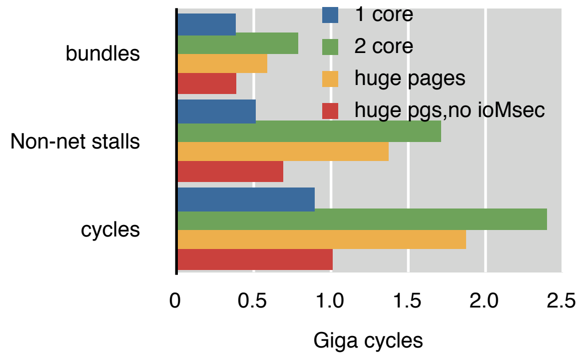

Figure 12. Savings w/ huge pages, no ioMsecs

The top group of bars measures instruction bundles retired, and shows that each of the two optimizations contributed about equally to reduce the number of instructions in the 2-core case back to rough equality with the one-core case. The second group of bars reports on the stalls, and shows that eliminating the "ioMsec" calls was a greater effect in reducing these stalls. More stalls remained in the dual-core case even after both optimizations, but we still had several deferred issues yet to deal with. The third group of bars shows total cycles, or actual time, and demonstrates the combined effect of bundles and stalls.

Returning to the world of real execution on hardware in real time, we measured the time to run the compiler benchmark. The optimizations we had implemented improved single-core performance as well as the other cases.

Much, though not all of the mystery had been solved: After fixing these two problems, the two-core vs. one-core ratio improved from about a factor of 3 to a factor of 1.7:

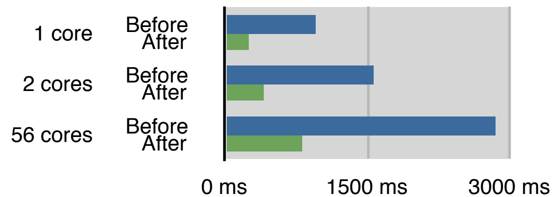

Figure 13. Effect of using huge pages and eliminating ioMsec calls

Returning to the TILE64 simulator, we took another look at stalls given our optimizations. (See Figure 14.) The three greatest sources of stalls for our current two-core system were L2 data, L1 instruction, and L2 instruction (TLB

miss, Napping and L2 busy stalls were insignificant and thus elided from the chart). At this point, we could have gone after L1 instruction stalls, but those were not relevant to the extra cores. Instead, it was the L2 data stalls which increased the most with the addition of a second core:

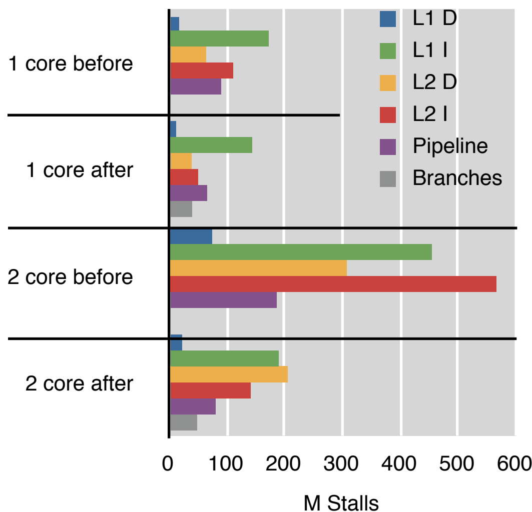

Figure 14. Stalls before and after

The simulator had pointed us to fetching bytecodes out of methods as a prime cause of these stalls. Since Smalltalk method objects are immutable, we started wondering if there were not some way to allow the hardware to replicate these and other immutable or even rarely-written objects (e.g. classes) among the cores as needed. It was time to tackle the previously deferred issues.

5.5. The Read-Mostly Heap

Many systems contain objects that are read far more frequently than written. In Smalltalk/Squeak, the currently executing method is read for every bytecode and literal interpreted, but it is never written; the class of a message receiver is read for every method lookup, but only written when a class-instance variable is changed (class modifications are performed with a copy-on-write scheme); and a class's method dictionary is read for every method lookup, but only changed when a method is added, modified, or removed. In addition, many application-level objects exhibit periods in which reads vastly outnumber writes. For instance, a Smalltalk Point object has its x and y variables set upon initialization and is almost never mutated thereafter.

Replication can save time if the read frequency far exceeds the write frequency. Although the default caching regime for user data on the TILE64 platform maintains coherency by confining the set of cores that may cache a line to a single core, there is an alternative: a *user-managed* regime. When a page is operating under this regime, any cache line may be cached on as many cores as needed. However, unlike a processor with a handful of cores providing memory coherence in hardware, the time savings accrued when reading data residing in a local cache must be paid for with a substantial increase in the time required to modify the data. In order to modify a user-managed cache line, the application must first force every other core to invalidate any cached copies of the data, then must write the data, and finally must force the cache line out to main memory before any other core attempts to read the data. Consequently, the user-managed memory policy optimizes reads at a great cost in both time and complexity for writes. Tables 2 and 3 compare read and write operations for the two regimes.

5.5.1. Read-Only, or Read-Mostly

If write operations are to be expensive and complicated, perhaps only immutable objects such as compiled methods should be placed in user-managed memory. Such a design would simplify our system at the cost of missing opportunities to replicate other objects such as classes and method dictionaries. If our system were to allow these read-mostly objects to occupy user-managed memory, a store barrier would be required in order to instruct the other cores to invalidate a line before a given core performed a store to it. A decision had to made that would trade off performance against simplicity.

	user-managed regime
read	2 cycles if in L1, 8 cycles if in L2
write	1. other tiles must invalidate the cache line, 2. storing tile stores the data, 3. storing tile flushes the line, 4. storing tile and performs a fence operation

Table 2. The TILE64 User-Managed Regime

read-wriite regime		
	home time	non-home tile
read	2 cycles if in L1, 8 cycles if in L2	~ 40 cycles if in home L2, ~ 80 cycles if in memory
write	hardware does it	hardware does it

Table 3. The TILE64 Read-Write Regime

Sometimes it's easy; we discovered that even without a store barrier, the compiler benchmark would run with all (non-context) objects placed in read-mostly memory, a fortuitous and fortunate happenstance that perhaps sheds light on the difference between benchmarks and real workloads. So, we performed a quick-and-dirty experiment: We ran the benchmark on one, two, and 56 cores, with no replication (all objects in read-write memory), method-only replication, and universal replication:

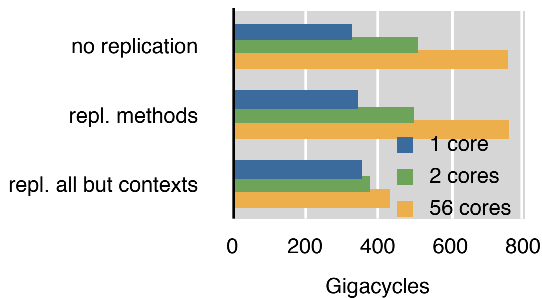

Figure 15. Quick & Dirty Experiments / no coherence code

The first set of bars shows the system without replication: The two core configuration was perhaps 1.5 times slower than the single-core, and the 56 core configuration is between two and three times slower. The second set of bars showed the results for employing the user-managed policy for methods only: Surprisingly, there was little improvement. Applying the user-managed policy to all objects showed much better scalability. This experiment convinced us to shoulder the burden of a store barrier in order to apply the user-managed regime to such read-mostly objects.

5.5.2. Incorporating User-Managed Memory

Our first design to incorporate user-managed memory added another heap, dubbed the "read-mostly heap" to our memory system:[1]

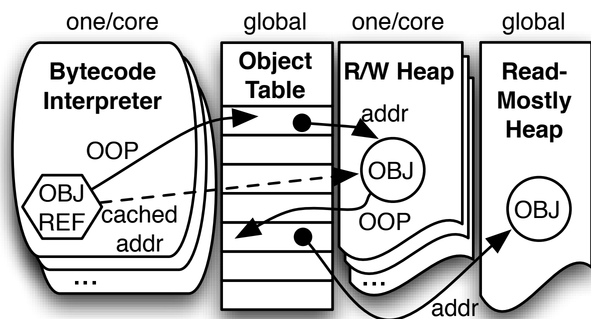

Figure 16. Adding a global read-mostly heap

A store barrier intercepts writes to objects and performs the extra work to ensure coherence if the object resides in this read-mostly heap. For virtual-machine-level operations that modify objects in bulk, such as garbage collection, the extra coherency-preserving work is performed once for the whole read-mostly heap. This mechanism would have been more difficult had we also been tackling application-level multithreading: What happens if every core decides to simultaneously ask every other core to invalidate a cache line? That question is reserved for the future.

With this addition of the read-mostly heap the baton-passing algorithm became slightly more elaborate: If the receiver of a message-send is in the read-mostly heap, the baton need never be passed, since any core is as good as any other.

Our design had to include policies and mechanisms to attempt to put the appropriate objects in the read-mostly heap. New objects and contexts were assumed to exhibit a high mutation frequency for initialization and interpretation, and were thus allocated in the read-write heap. In fact, contexts were never allowed in the read-mostly heap.

Upon performing a store into any object residing in the read-mostly heap, the virtual machine would move the object to the read-write heap in order to forestall catastrophic performance degradation in case a series of stores were about to be performed on that object.

As it seemed too burdensome to automatically discover objects that had not been modified for a while, the responsibility for moving a new object into the read-mostly heap was placed on the application by providing primitive operations that would move either one or all (non-context) objects to the read-mostly heap. We assumed that the application code would periodically move everything to the read-mostly heap and let the virtual machine weed out the mutations, and/or initialization methods would be augmented to move the newly initialized instance into the read-mostly heap. An object read from the snapshot was assumed to be stable and placed into the read-mostly heap (see figure 17).

When we measured this system, the numbers looked too good—the dual core time was only 1.1 times as slow as the single core—and we realized that the baton was almost never getting passed! Since most objects were in the single read-mostly heap there was rarely a need to move the mutator thread. This problem was remedied by splitting up the global read-mostly heap into per-core pieces (powers of two in size) so that, even though it made no difference to the hardware, our system could use an object's address as a means to indicate which core should run the interpreter when sending a message to that object. In addition, we added a per-object flag to *disable* baton-passing in order to retain the older behavior when desired. Figure 18 shows the revised baton-passing algorithm, where "wants baton" means "does not have the don't-pass bit set."

[1] Actually, we initially named this heap the "incoherent" heap, but nobody understood what we were talking about. Our thanks go to David Bacon for pointing out the obscurity of that nomenclature.

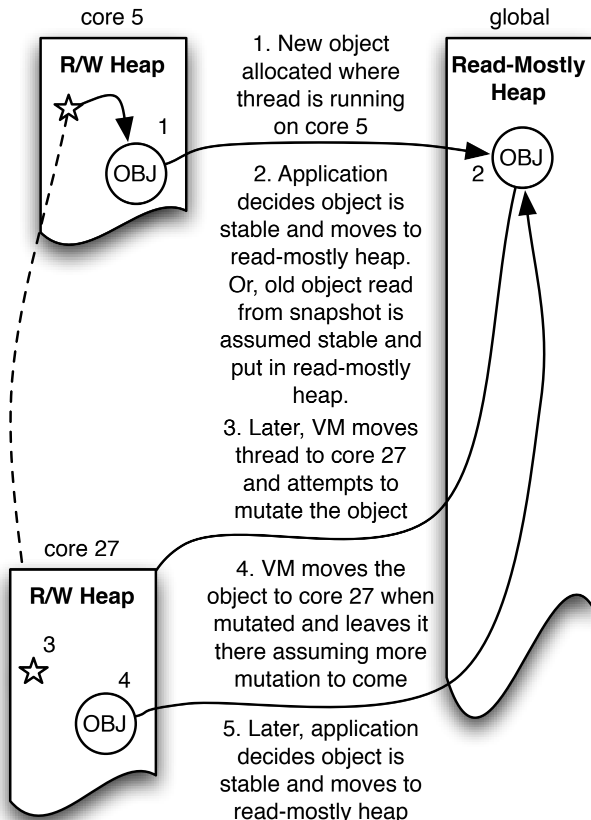

Figure 17. Heap local lifecycle

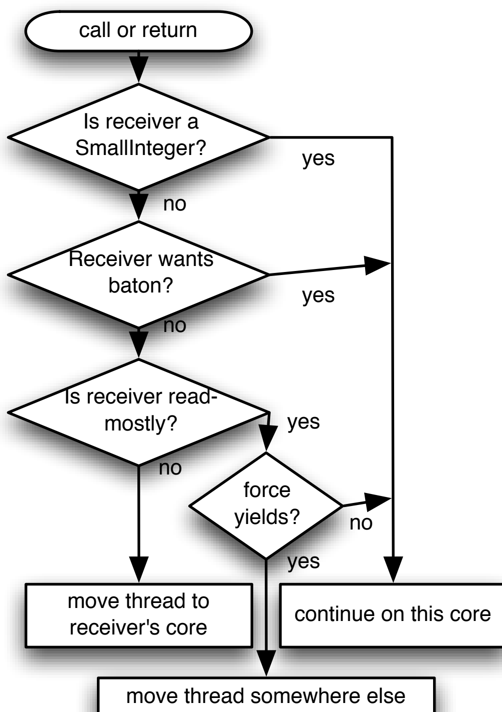

Figure 18. Final baton passing algorithm

Conclusions and Future Work

In adapting a large, centralized heap of many small objects to a collection of heaps distributed over a multiplicity of medium-sized caches, our design evolved in stages. Starting with a reference implementation, we

- introduced distinct types for references vs. addresses,
- shared the address space across all cores to facilitate inter-core object references,
- added an object table to support object migration,
- used ordinary objects for activation records to facilitate thread migration,
- separated the global object heap into per-core heaps,
- enforced locality for receiver and activation record access by passing the baton to the receiver's core when invoking a virtual function and by allocating the activation record on the same core as the receiver,
- added a global heap with user-managed caching in order to replicate read-mostly objects,
- implemented a store barrier in order to maintain coherence for the read-mostly heap, and
- separated the global read-mostly heaps into per-core heaps.

Each stage ideally followed a sequence of design and implementation, measurement, interpretation and evaluation, optimization, and again, measurement. Even without parallel workloads, efficiency has been a challenge at every stage. Because the overall goal of the project was the programming model, and given budget constraints, we did not get every answer we wanted before moving on.

What lessons can we learn? One obvious but important lesson is that manycore systems are very different platforms, offering far faster communication than distributed systems, far smaller caches and TLBs than modern few-core systems, easier address-space sharing than distributed systems, but more problematic cache coherency than few-core systems. This last issue is widely recognized as a major challenge in designing and exploiting manycore architectures. For example, Tilera further addresses cache coherency in their second-generation chip, the TILEPro64 [22].

With a manycore hardware architecture, the virtual machine designer faces different trade-offs than with traditional architectures. For example, migrating objects to achieve locality takes less time than on a distributed system, yet saves much more time than on a few-core system. We observed a bit of this difference when we implemented the read-mostly heap and allowed many objects to be not only migrated to where they were needed, but replicated as well.

Now that we have a stable manycore object memory model, our focus has shifted to implementing application-level multithreading to complete our experimental platform for prototyping programming paradigms and concepts. We will revisit optimization when our system is running parallel workloads.

Acknowledgements

The authors would like to thank our IBM colleagues Mark Wegman, Erik Altman, Michael Hind, David Bacon and David Grove for many insightful contributions to this effort; the Squeak community for its passionate advancement and preservation of the original Smalltalk IDE; Leo Ungar for his editing; and Richard Schooler, VP SW Engineering at Tilera, and his team for their excellent support during this project.

References

1. IBM, "The IBM® Blue Gene®/P Solution," 2009; http://www-03.ibm.com/systems/deepcomputing/bluegene/.
2. IBM, "IBM Cell Broadband Engine technology," 2009; http://www-03.ibm.com/technology/cell/index.html.
3. Tilera Corp., "Tilera," 2009; http://www.tilera.com.
4. Squeak.org, "Squeak," 2009; http://squeak.org/.
5. J. Pallas and D. Ungar, "Multiprocessor Smalltalk: a case study of a multiprocessor-based programming environment," PLDI, 1988.
6. J.I. Pallas, "Multiprocessor Smalltalk: implementation, performance, and analysis," Computer Science, Stanford University, Stanford, CA, 1990.
7. C.P. Thacker and L.C. Stewart, "Firefly: A multiprocessor workstation," ASPLOS II, 1987.
8. D.M. Ungar and D.A. Patterson, "Berkeley Smalltalk: Who knows where the time goes?," *Smalltalk-80: History, Words of Advice*, G. Krasner, ed., Addison-Wesley, 1983, pp. 189-206.
9. D. Ungar, "Generation Scavenging: A non-disruptive high performance storage reclamation algorithm," Software Engineering Symposium on Practical Software Development Environments, 1984.
10. D.M. Ungar, *The Design and Evaluation of a High Performance Smalltalk System*, MIT Press, 1987, p. 250.
11. MIT Artificial Intelligence Laboratory, "The Jellybean Machine," 1998; http://cva.stanford.edu/projects/j-machine/.
12. W. Horvat, *Concurrent Smalltalk on the Message-Driven Processor*, MIT Computer Science and Artificial Intelligence Lab, 1991; http://dspace.mit.edu/handle/1721.1/7090.
13. I. Williams, "The Mushroom Machine - An Architecture for Symbolic Processing," IEE Colloquium on VLSI and Architectures for Symbolic Processing, 1989.
14. M. Wolczko and I. Williams, "The influence of the object-oriented language model on a supporting architecture," 26th Hawaii Conference on System Science, 1994.
15. I. Williams and M. Wolczko, "An Object-Based Memory Architecture," Fourth International Workshop on Persistent Object Systems, 1991.
16. J. Stokes, "MIT startup raises multicore bar with new 64-core CPU," 2007.
17. Tilera Corp., *Multicore Development Environment - Product Brief*, T. Corp., 2008; http://tilera.com/pdf/ProductBrief_MDE_Web_v2.pdf.
18. D. Ingalls, et al., "Back to the future: The story of Squeak, A practical Smalltalk written in itself," OOPSLA, 1997; http://www.vpri.org/pdf/backto_TR-1997-001.pdf.
19. T. Reenskaug, "The Model-View-Controller (MVC). Its Past and Present," JAOO, 2003; http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf.
20. T. Reenskaug, *MODELS - VIEWS - CONTROLLERS*, 1979; http://folk.uio.no/trygver/1979/mvc-2/1979-12-MVC.pdf.
21. T. Reenskaug, "MVC Xerox PARC 1978-79," Trygve/MVC.
22. Tilera Corp., *TileraPro64 Processor - Product Brief*, T. Corp., 2008; http://tilera.com/pdf/ProductBrief_TILEPro64_Web_v2.pdf.